

# A massively parallel GPU-accelerated model for analysis of fully nonlinear free surface waves

**Engsig-Karup, Allan Peter; Madsen, Morten G.; Glimberg, Stefan Lemvig**

*Published in:*
International Journal for Numerical Methods in Fluids

*Link to article, DOI:*
10.1002/fld.2675

*Publication date:*
2011

*Document Version*
Publisher's PDF, also known as Version of record

INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS
*Int. J. Numer. Meth. Fluids* 2012; **70**:20–36
Published online 6 Sept 2011 in Wiley Online Library (wileyonlinelibrary.com/journal/nmf). DOI: 10.1002/fld.2675

# A massively parallel GPU-accelerated model for analysis of fully nonlinear free surface waves

A. P. Engsig-Karup*,†, Morten G. Madsen and Stefan L. Glimberg

*Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark*

SUMMARY

We implement and evaluate a massively parallel and scalable algorithm based on a multigrid preconditioned Defect Correction method for the simulation of fully nonlinear free surface flows. The simulations are based on a potential model that describes wave propagation over uneven bottoms in three space dimensions and is useful for fast analysis and prediction purposes in coastal and offshore engineering. A dedicated numerical model based on the proposed algorithm is executed in parallel by utilizing affordable modern special purpose graphics processing unit (GPU). The model is based on a low-storage flexible-order accurate finite difference method that is known to be efficient and scalable on a CPU core (single thread). To achieve parallel performance of the relatively complex numerical model, we investigate a new trend in high-performance computing where many-core GPUs are utilized as high-throughput co-processors to the CPU. We describe and demonstrate how this approach makes it possible to do fast desktop computations for large nonlinear wave problems in numerical wave tanks (NWTs) with close to 50/100 million total grid points in double/single precision with 4 GB global device memory available. A new code base has been developed in C++ and compute unified device architecture C and is found to improve the runtime more than an order in magnitude in double precision arithmetic for the same accuracy over an existing CPU (single thread) Fortran 90 code when executed on a single modern GPU. These significant improvements are achieved by carefully implementing the algorithm to minimize data-transfer and take advantage of the massive multi-threading capability of the GPU device. Copyright © 2011 John Wiley & Sons, Ltd.

Received 13 April 2011; Revised 28 July 2011; Accepted 31 July 2011

KEY WORDS: nonlinear water waves; coastal and offshore engineering; finite difference method; potential flow; time domain; scientific GPU computations; high-performance computing

## 1. INTRODUCTION

### *1.1. Motivation*

In coastal engineering, wave models are used by engineers for the design of human made structures and predictive purposes for estimating flow kinematics and loads. For practical applications in coastal engineering, non-dispersive shallow water models and weakly dispersive boussinesq-type are in widespread use because software based on such models are relatively easily made, both efficient and robust. In offshore and ocean engineering, these models are less attractive because they fail to accurately account for dispersive properties of wave propagation at intermediate or deep waters because of the analytical simplifications introduced in the derivation procedures. Although, full potential flow theory in three space dimensions is a useful modeling basis in both shallow and deep waters, it seems to be still widely believed in the community of practitioners that this modeling basis is *"far too heavy"* [1] and *"it is well known that full methods in 3D are slow"* [2] and *"Although*

*Correspondence to: A. P. Engsig-Karup, Department of Informatics and Mathematical Modeling, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark.

†E-mail: apek@imm.dtu.dk

*developing a three-dimensional numerical model from the Laplace equation is straightforward, it has little use in coastal engineering practice (as a result of its exorbitant memory requirements which limit its application to very small domains"* [3] compared with the above-mentioned widely used alternatives.

With recent developments in a finite difference algorithm for solving nonlinear free surface potential flow problems in three space dimensions [4,5] efficiently together with improvements in modern hardware, we seek to demonstrate that it is now possible to significantly reduce the barriers for practical use of full potential flow theory as the modeling basis for efficient solution of coastal and offshore engineering problems. Our strategy is to do *proof-of-concept* by utilizing modern graphics processing units (GPUs) for massively parallel computation using a heterogenous CPU-GPU hardware setup that constitutes what can be considered as an affordable *standard* consumer desktop environment.

Algorithms that map well to parallel programming models for current GPUs can potentially be utilized for significant acceleration of applications for three main reasons, namely, high on-chip bandwidth, many cores suitable for data processing in parallel and effective latency hiding through massive multi-threading capability. The utilization of GPUs in coastal engineering is (still) not in widespread use, however, with the current trends and progression speed in new developments of hardware for scientific computations, one might be tempted to predict a change in the near future, because many application within coastal engineering should map well to GPUs, for example, models based on the long wave assumptions such as shallow water and Boussinesq-type applications discretized using explicit algorithms. In addition, the scalable parallel programming model developed for GPUs will likely also be applicable for future multi-core CPUs as discussed by [6].

### *1.2. Background*

Coastal and offshore engineering problems are in general very compute-intensive and as of today, the solution of problems at very large scales are not as common (e.g., see [1]) despite that massive parallel computing has long been considered an important technology (e.g., see [7, 8]). Thus, a key challenge is to port existing code, analyze, improve and redesign algorithms to fully utilize available hardware resources to maximize computational throughput.

Computational hardware are being continuously improved every year and will continue to do so for many decades to come. However, with new trends in high-performance computing where typically multi-core and many-core technologies are available in standard consumer hardware, we have entered a new era in computation [9]. These developments in computational resources improve the basis for solving increasingly larger and more complex problems. Soon multi-core CPUs with tens of cores (consisting of scalar processing units) will be standard and with even more cores in the future. Mass-produced GPUs already contains hundreds of scalar processing units, and therefore enable massively parallel computing today.

With the new trends in hardware developments and programming tools for general purpose computing, it is now accepted that it is no longer an option to just wait for faster hardware to get better performance [10] or to buy a very large and expensive cluster of processing units when the basic algorithms are not scalable to arbitrary numbers of cores. Many existing codes based on single core computations need to be entirely rewritten to keep up with the developments in hardware to address new opportunities for exploiting thousands of processing cores working in parallel. Although new hardware offers much more computational power in the form of computational parallelism, a key challenge is still to overcome what is known as the *memory wall* [11]. The increasing gap existing between computational versus memory-transfer throughput requires hardware that makes it possible to hide memory latency and favors algorithmic methods with a low data-transfer relative to computations (communication costs). For applications that demand high memory bandwidth, it is possible to use accelerators such as GPUs, CELL processors, and field programmable gate arrays. In this work, we choose to focus on the use of GPUs because of the software tools made available for programming such hardware.

Modern GPUs are based on a single-instruction multiple-thread architecture. The use of special purpose GPUs for acceleration of general purpose scientific computations have increased,



DOI: 10.1002/fld

significantly, the last few years, and the state-of-the-art is rapidly developing, for example, see [12–14]. Main reasons for this successful turnaround are that these accelerator devices are designed as *special purpose* co-processors for the use with current *general purpose* CPUs, they are programmable, mass produced, and therefore offered at affordable prices and already present in many machines. Furthermore, these compute devices support high computational density operations, on-chip memory bandwidth is high, hardware support for spawning and switching between thousands of threads for hiding memory latency and maintaining effective instruction throughput. The barrier to exploit the hardware has been significantly reduced by the general purpose scalable parallel programming models (e.g., see [15]) and frameworks supported and provided by the hardware vendors, for example, the industry standards OpenCL and DirectCompute, or the vendor-specific compute unified device architecture (CUDA) by [16, 17], with the latter currently most widely adopted. These programming interfaces can be used by adding C extensions to existing code.

### *1.3. Paper contribution*

We are the first to present a massively parallel GPU implementation of a flexible-order finite difference model for fully nonlinear free surface flow based on direct solution of the Laplace problem in three space dimensions. The implemented algorithmic strategy has recently been proposed by [5] and can be considered as a generalization of the multigrid-based strategy first proposed by [18]. We demonstrate that the implemented model enables fast execution and analysis on heterogenous CPU–GPU hardware systems, for example, in standard desktop consumer hardware. We present a basic performance analysis and results hereof and discuss when the proposed wave model can be used for *real-time* analysis.

### *1.4. Paper organization*

The paper is organized as follows. Following [4], we present the governing equations in Section 2 and summarize the numerical methods for discretization, hereof, in Section 3. In Section 4, we propose to use the low-storage, efficient, and scalable algorithm developed by [5] for massively parallel computations for the solution of the governing equations. In Section 5, we detail a novel massively parallel implementation for heterogenous CPU–GPU hardware. Section 6 contains a parallel performance analysis of our implementation.

## 2. FULLY NONLINEAR FREE SURFACE POTENTIAL FLOW MODEL

A model for fully nonlinear free-surface potential flow above uneven bottoms is described briefly. The model is derived on assuming that viscous and turbulence effects of the fluid flow can be neglected and that waves are nonbreaking. These assumptions are reasonable for many applications in coastal and offshore engineering. In the following, we consider a setup for what is referred to as a numerical wave tank (NWT).

A Cartesian coordinate system $(x, y, z) = (\mathbf{x}, z)$ is adopted with the $xy$-plane located at the still water level and the $z$-axis pointing upwards. The still water depth is defined by $z = -h(\mathbf{x})$ and the position of the free surface by $z = \zeta(\mathbf{x}, t)$. The magnitude of the gravitational acceleration $g$ in the vertical is assumed constant. Assuming an inviscid fluid and an irrotational flow, the fluid velocity $(u, v, w) = (\nabla, \partial_z)\phi$ is defined by the gradient of a scalar velocity potential $\phi(\mathbf{x}, z, t)$, where $\nabla = (\partial_x, \partial_y)$ is the horizontal gradient operator and $\partial_x$ denotes partial differentiation with respect to the $x$-coordinate.

Following [19], the evolution of the free surface in an Eulerian frame of reference can be expressed in terms of the kinematic and dynamic boundary conditions in the form

$$\partial_t \zeta = -\nabla\zeta \cdot \nabla\tilde{\phi} + \tilde{w}(1 + \nabla\zeta \cdot \nabla\zeta), \tag{1a}$$

$$\partial_t \tilde{\phi} = -g\zeta - \tfrac{1}{2}\left(\nabla\tilde{\phi} \cdot \nabla\tilde{\phi} - \tilde{w}^2(1 + \nabla\zeta \cdot \nabla\zeta)\right), \tag{1b}$$



DOI: 10.1002/fld

in terms of the free surface quantities $\tilde{\phi} = \phi(\mathbf{x}, \zeta, t)$ and $\tilde{w} = \partial_z \phi|_{z=\zeta}$. To find $\tilde{w}$ and integrate these equations forward in time requires solving the Laplace equation in the fluid volume $\Omega$. A well-posed Laplace problem is achieved by specifying known $\tilde{\phi}$ at the free surface, together with kinematic boundary conditions at the vertical sides and at the bottom of the NWT

$$\phi = \tilde{\phi}, \quad z = \zeta, \tag{2a}$$

$$\nabla^2 \phi + \partial_{zz}\phi = 0, \quad -h \leqslant z < \zeta, \tag{2b}$$

$$\mathbf{n} \cdot (\nabla, \partial_z)\phi = 0, \quad (\mathbf{x}, z) \in \partial\Omega \tag{2c}$$

where $\boldsymbol{n} = (n_x, n_y, n_z)$ is an outward pointing normal vector to the solid domain boundary surfaces $\partial\Omega$. The free surface is a time-dependent moving boundary with an *a priori* unknown position practically resulting in a time-dependent domain. By employing a $\sigma$-transform in the vertical coordinate as proposed by [18], the numerical operations can be done in a *fixed* computational domain. The following change of variable in the vertical direction is introduced

$$\sigma \equiv \frac{z + h(\boldsymbol{x})}{d(\boldsymbol{x}, t)}, \quad 0 \leqslant \sigma \leqslant 1 \tag{3}$$

where $d(\boldsymbol{x}, t) = \zeta(\boldsymbol{x}, t) + h(\boldsymbol{x})$ is the height of the ware column. The fluid domain is mapped to a time-invariant computational domain where the Laplace problem (2) is expressed in the transformed coordinate system as

$$\Phi = \tilde{\phi}, \quad \sigma = 1, \tag{4a}$$

$$\nabla^2\Phi + \nabla^2\sigma(\partial_\sigma\Phi) + 2\nabla\sigma \cdot \nabla(\partial_\sigma\Phi) + (\nabla\sigma \cdot \nabla\sigma + (\partial_z\sigma)^2)\partial_{\sigma\sigma}\Phi = 0, \quad 0 \leqslant \sigma < 1, \tag{4b}$$

$$\mathbf{n} \cdot (\nabla, \partial_z\sigma\partial_\sigma)\phi = 0, \quad (\mathbf{x}, \sigma) \in \partial\Omega \tag{4c}$$

where the scalar velocity function $\Phi(\boldsymbol{x}, \sigma, t) = \phi(\boldsymbol{x}, z, t)$ contains all information about the flow kinematics in the fluid volume. The derivatives of the coordinate $\sigma$ are evaluated as nonlinear coefficients in (4) and can be expressed as

$$\nabla\sigma = \tfrac{1-\sigma}{d}\nabla h - \tfrac{\sigma}{d}\nabla\zeta, \tag{5a}$$

$$\nabla^2\sigma = \tfrac{1-\sigma}{d}\left(\nabla^2 h - \tfrac{\nabla h \cdot \nabla h}{d}\right) - \tfrac{\sigma}{d}\left(\nabla^2\zeta - \tfrac{\nabla\zeta \cdot \nabla\zeta}{d}\right) - \tfrac{1-2\sigma}{d^2}\nabla h \cdot \nabla\zeta - \tfrac{\nabla\sigma}{d} \cdot (\nabla h + \nabla\zeta), \tag{5b}$$

$$\partial_z\sigma = \tfrac{1}{d}. \tag{5c}$$

All of these coefficients can be computed explicitly from the known two-dimensional free surface and bottom positions at given instants of time.

Furthermore, when $\Phi$ is known, the vertical velocity can be determined from

$$w = \partial_z\phi = \partial_z\sigma\partial_\sigma\Phi. \tag{6}$$

## 3. NUMERICAL METHOD FOR DISCRETIZATION

The solution of the governing equations given in Section 2 and application to several wave problems have been done using an efficient and fairly optimized single core CPU (single thread) implementation. For complete analysis and experimental validation of the algorithm and the involved iterative strategy, we refer to [4, 5, 20]. The discretization approach based on a flexible-order finite difference method and is briefly summarized in the following for the setup of a NWT. This approach has been demonstrated to be robust, efficient, and accurate and thereby enable numerical solution of a broad range of important practical applications.

A method of lines approach is used for the discretization of the governing equations stated in Section 2. For the time-integration of the free-surface conditions (1), a classical explicit four-stage, fourth-order Runge–Kutta (ERK4) scheme is employed as it is not subject to any severe stability constraints on the choice of time steps and is straightforward to parallelize. For the spatial discretization, a free surface grid consisting of $(N_x, N_y)$ points is defined along the horizontal



DOI: 10.1002/fld

$xy$-axes at which the free surface variables $\zeta$ and $\tilde{\phi}$ are to be evolved. At the structural boundaries of the domain, that is, at the bottom and wall sides, no normal flux conditions (4c) are imposed using a stencil half-width layer of fictitious ghost points in the horizontal and only one layer below the bottom.

For the discretization of the transformed Laplace problem (4), $N_z$ points are defined in the vertical direction below each horizontal free surface grid point, which can be arbitrarily spaced between $0 \leqslant \sigma \leqslant 1$. Choosing $r$ nearby points in the $x$-direction, for example, $\alpha_l$ left-most and $\alpha_r$ right-most neighboring values, allows finite difference schemes for the one-dimensional first and second derivatives in $(x, y, \sigma)$ to be developed at each of the $x$, $y$, and $\sigma$ positions on the grid. A $p$'th order accurate approximation for the $q$th derivative can be expressed in the general form

$$\left.\frac{\partial^q u}{\partial x^q}\right|_{x=x_i} = \sum_{n=-\alpha_l}^{\alpha_r} c_n^x u_{i+n} + \mathcal{O}(\Delta x^p) \tag{7}$$

with $c_i^x$ being the finite difference weights determined by the order of differentiation and the computational grid distances between points in the $x$-direction. Mixed $x\sigma$-derivatives and $y\sigma$-derivatives can be computed from the one-dimensional stencils by tensor products, for example,

$$\left.\frac{\partial^2 u}{\partial x \partial \sigma}\right|_{(x,\sigma)=(x_i,\sigma_i)} = \sum_{m=-\alpha_l}^{\alpha_r} \sum_{n=-\gamma_l}^{\gamma_r} c_m^x c_n^\sigma u_{i+m,j+n} + \mathcal{O}(\Delta x^p) \tag{8}$$

where $\gamma_l$ and $\gamma_r$ are responsible for indexing the neighboring values in the vertical. This will result in approximations that are formally order $(r-1)$ for centered stencils and $(r-q)$ for off-centered stencils with $q$ the order of the derivative. The order of the spatial discretization schemes is kept flexible to have two convergence strategies available for combinations, namely, $h$-adaptivity and $p$-adaptivity where either the spatial resolution or the order of the scheme is increased, respectively. The resulting discrete Laplace problem can be stated as a rank $n = N_x \cdot N_y \cdot N_z$ linear system of equations,

$$\mathcal{A}\boldsymbol{\Phi} = \mathbf{b}, \tag{9}$$

where $\mathcal{A}$ is in general a large sparse non-symmetric matrix, $\boldsymbol{\Phi}$ is vector of values for the unknown scalar velocity potential, and $\mathbf{b}$ is a vector accounting for in-homogenous boundary conditions.

## 4. PARALLEL ALGORITHM

The main goal is to implement a massively parallel and scalable algorithm for solving large nonlinear water wave problems, efficiently, on modern many-core hardware. To achieve this goal, two choices have to be made, namely, what hardware is to be used and how to implement the algorithm to utilize the available hardware efficiently.

For algorithms to be useful in practice for large problems, essential requirements are correctness, robustness, efficiency, and scalability as described by [21,22]. The fully nonlinear free surface wave problem in three space dimensions can be solved using the recently developed algorithmic approach summarized in Section 3, which fulfills these fundamental properties. In this work, a main concern is to maximize the ability to solve large problems, and therefore the storage space requirements of the algorithm is a determining factor in the choice of suitable methods.

In the recent work of [5], an efficient explicit algorithm based on a low-storage multigrid preconditioned defect correction (DC) method for the solution of the Laplace problem is proposed. This algorithmic approach is a promising strategy for parallel computations as it is explicit, the memory requirements are minimal and few global synchronization points are needed.

The free surface problem (1) is essentially one spatial dimension (i.e., $\mathcal{O}(N_z)$) smaller than the Laplace problem (2). Consequently, for large problems, the free surface problem requires much less effort to advance in time, for example, using ERK4, in comparison with the iterative solution of the Laplace problem (2). Thus, the computational bottleneck problem is the solution of the linear



DOI: 10.1002/fld

system (9) every time step of the time-integration of the evolution of the free surface problem. This linear system can be solved efficiently starting from some initial guess $\Phi^{[0]}$ by a DC method with iterations given in compact form

$$\mathbf{\Phi}^{[k+1]} = \mathbf{\Phi}^{[k]} + \delta^{[k]}, \quad \delta^{[k]} = \mathcal{M}^{-1}(\mathbf{b} - \mathcal{A}\mathbf{\Phi}^{[k]}), \quad k = 0, 1, 2, ... \tag{10}$$

where $\mathcal{M} \approx \mathcal{A}$ can conceptually be considered as (the action of) a preconditioning matrix responsible for minimizing de-acceleration of convergence over the single-iteration scheme achieved if $\mathcal{M} \equiv \mathcal{A}$. When a full high-order discretization is employed for the discretization of (9) and a reduced-order multigrid method is employed for determining the correction $\delta^{[k]}$ in the preconditioning step in (10), the resulting method can be perceived as a $p$-multigrid method and is given in pseudocode in Algorithm 1.

In this work, we employ a second-order multigrid discretization preconditioning strategy based on a time-constant discretization of the linearized system matrix (assume $\zeta \approx 0$ in (10)) which was demonstrated to be efficient by [5]. With the resulting DC method based on a two-step recurrence combined with the same multigrid preconditioning strategy proposed by [4], the storage requirements can be kept minimal (see also discussion in Section 5.4). The best preconditioning strategy was found to be a multigrid method based on a Red-Black Zebra-Line Gauss–Seidel (RB-ZL-GS) smoothing strategy combined with semi-coarsening. The multigrid strategy MG-RB-ZL-GS-1V(1,1) based on a V-cycle with one pre-smoothing and post-smoothing was found to be efficient for the entire application range from shallow to deep water in the current numerical model. This is in contrast to point-based smoothing methods that exhibit slow convergence when discrete anisotropy is high (fx. in shallow water). We, therefore, only consider this strategy in the following.

**Algorithm 1** Defect Correction Method for approximate solution of $\mathcal{A}\mathbf{\Phi} = \mathbf{b}$

**1** Choose $\Phi^{[0]}$ /* initial guess */
**2** $k = 0$
**3 repeat**
**4** $r^{[k]} = b - \mathcal{A}\Phi^{[k]}$ /* high-order defect */
**5** Solve $\mathcal{M}\delta^{[k]} = 5r^{[k]}$ /* preconditioning problem */
**6** $\Phi^{[k+1]} = \Phi^{[k]} + \delta^{[k]}$ /* defect correction */
**7** $k = k + 1$
**8 until** $(||r^{[k]}|| < ||r^{[0]}|| \cdot rtol + atol)$ *and* $(k < maxiter)$;

Geometric multigrid methods are multi-level methods [23] and can be constructed from three basic components, namely, a fine-to-coarse grid (restriction) operator, a coarse-to-fine (prolongation) operator, and a smoothing (relaxation) operator. With second-order discretizations employed, it is sufficient to use tri-linear interpolation for the adjoint grid-transfer operations. The remaining essential component is the smoother.

For each of the $N_x \cdot N_y$ the free surface grid points, the RB-ZL-GS smoothing strategy requires solving a small $m \times m$ linear system of equations (with $m = N_z$). We employ a discretization of the transformed Laplace problem with a single layer of fictitious ghost points below the bottom for imposing kinematic boundary conditions. Then these small linear systems have a general and nearly tri-diagonal structure of the form

$$\begin{bmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & a_{m-1} & b_{m-1} & c_{m-1} \\ 0 & & e_m & a_m & b_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_m \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_m \end{bmatrix} \tag{11}$$

This system can be solved directly using a modified Thomas Algorithm (TA) that takes into account an extra nonzero coefficient denoted $e_m$. This changes the standard TA to a forward



DOI: 10.1002/fld

elimination phase resulting in modified coefficients for the system

$$c_i' = \begin{cases} \dfrac{c_1}{b_1} & ,i = 1 \\ \dfrac{c_i}{b_i - c_{i-1}' a_i} & ,i = 2,3,...,m-1 \end{cases} \tag{12}$$

and

$$d_i' = \begin{cases} \dfrac{d_1}{b_1} & ,i = 1 \\ \dfrac{d_i - d_{i-1}' a_i}{b_i - c_{i-1}' a_i} & ,i = 2,3,...,m-1 \\ \dfrac{(d_i - e_m d_{i-2}') - d_{i-1}'(a_i - e_m c_{i-2}')}{b_i - c_{i-1}'(a_i - e_m c_{i-2}')} & ,i = m \end{cases} \tag{13}$$

The solution can then be obtained by back substitution as

$$x_m = d_m', \quad x_i = d_i' - c_i' x_{i+1}, \quad i = m-1, m-2, ..., 1. \tag{14}$$

This modified TA is inherently sequential and can be implemented with memory requirements of complexity $2m$. This is possible when we take advantage of the known time-constant sparsity structure of the small system matrix that results from the discretization in the vertical of the linearized version of (4).

The analysis of the finite difference model by [4, 20] demonstrates that $N_z \approx 5$–$10$ points in the vertical are sufficient to achieve satisfactory engineering accuracy levels of, say, less than two percent in wave dispersion and kinematics for typical simulations in costal engineering. This implies that, for typical simulations, the block systems of the form (11) that arise in Zebra-Line smoothing operations will be small.

At the coarsest grid level in the multigrid procedure, the number of required smoothing operations depends on the grid size and should ensure that the accuracy is comparable with the level of discretization errors to not affect overall convergence rate (e.g., see [24, 25]). To avoid significant processor idle time that may impact parallel efficiency (cf. Amdahl's law), our multigrid algorithm is coarsened to the smallest grid level where there is just enough threads to keep the stream processors busy. At this level, we use on the order of 10 smoothing operations to solve the residual equations approximately.

The solver for the entire problem is compromised by pre-processing, time-integration, and post-processing steps. For time-integration based on explicit Runge–Kutta methods, the following fundamental algorithmic steps for each stage in such ordinary differential equation (ODE) solvers are performed:

1. Using input for in-homogenous boundary conditions, for example, the current free-surface state $\tilde{\phi}$, solve the transformed Laplace problem (4) using a defect correction method (cf. Algorithm 1).

1b. In step 5 of Algorithm 1, a preconditioning problem needs to be solved.

2. Compute vertical velocity at free surface $\tilde{w}$ using (6).
3. From the last free-surface state at previous time step defined in terms of $\zeta$, $\tilde{\phi}$, and $\tilde{w}$, advance solution by evaluating the right hand side function of (1).
4. Repeat from Step 1 until algorithm completes.

The bulk of the work is in Step 3 because this part requires processing data for three space dimensions. Step 1b solves the defect equation using a geometric multigrid method based on semi-coarsening to resolve problems related to discrete anisotropy. After every grid transfer and every sweep with smoothers, the boundary conditions needs to be enforced by explicitly updating fictitious ghost point layers.



DOI: 10.1002/fld

## 5. PARALLEL IMPLEMENTATION ON HETEROGENOUS CPU–GPU HARDWARE

### *5.1. Hardware characteristics*

In this work, we focus on utilizing many-core GPUs, which on current top end GPU hardware have theoretical on-chip bandwidth up to 192 GB/s and theoretical peak computational power on the order of 1.5/0.75 TFlop in single/double precision and equipped with up to 6 GB GDDR5 RAM for one GPU. Interestingly, these modern GPUs can easily be equipped in a desktop computer and constitute a powerful co-processing resource for a standard CPU host acting as a task manager. The CPU is designed for general purpose computations and the GPU for data-parallel computations, and this combined approach enables a powerful strategy for solving problems more efficiently. If an algorithm is executed on the CPU and co-processed on the GPU, it is today standard to transfer data through a standard peripheral component interconnect express (PCIe) $\times 16$ Gen2 bus interface with bandwidth of a most 8 GB/s. Thus, the PCIe link represents a potential bandwidth bottleneck for applications with excessive data-transfer between host and device. This suggests that algorithms for standard sparse numerical linear algebra operations (such as those presented in Section 4), which are typically *memory bound* (i.e., limited by available memory bandwidth), it can be advantageous to exploit the high on-chip memory bandwidth on such GPU devices. This also suggests that computationally intensive computations should be kept on the device and that data-transfers should be minimized to reduce memory latency and optimized with respect to access patterns. Thus, the strategy in this work is to offload all computations and runtime data-transfers to a single GPU to avoid the bandwidth bottleneck of the slow off-chip bus interface.

### *5.2. Parallel programming of GPUs*

As a starting point for programming GPUs, we choose to use the CUDA C parallel programming model proposed by Nvidia Corporation (Santa Clara, California, USA). This programming model is at the time of the work more mature and similar to the OpenCL industry standard developed by the Khronos Group that may become a future standard model for programming heterogeneous hardware. The chosen programming model promotes the use and development of massively parallel algorithms for computation and data-processing. It has been developed to ease the development of general purpose applications that can benefit from parallel execution on Nvidia GPUs only and requires no graphics knowledge. To fully utilize the GPUs, knowledge about the hardware characteristics and in particular the memory hierarchy are essential. The hardware limits of a given CUDA architecture is defined in terms of the Compute Capability (CC) version. The programming model and best practices are described in technical reports by [16, 17] and it is possible to also use micro-benchmarking of the hardware architecture to provide additional insight to hardware characteristics for Nvidia GPUs not detailed in public as investigated by [26].

To implement a parallel program on GPUs, the current programming paradigms require that the programmer decompose the program code into subroutines (tasks) that can be sequentially scheduled for execution on accelerator devices by the CPU host. Subroutines that are compiled for and executed on the GPU are referred to as *kernels* and should be primarily used for massively data-parallel (single instruction, multiple data ) tasks. Such kernels require an execution configuration that defines a virtual grid of threads. Each thread in the grid will process the kernel instructions and parts of the data, concurrently. The choice of kernel execution configurations is important for overall performance and can be differently optimized for different kernels. The optimal configuration is dependent on the scarce local memory requirements and physical limits of the architecture. Data are provided for kernels by transferring data from the host to the device memory from which they are processed by kernels in single program multiple data style. When kernels finish, a new kernel can be initiated at each stream processor.

### *5.3. Algorithmic components*

The costs of algorithms are fundamentally arithmetic operations (computation) and communication (data-transfer) requirements. By inspecting the entire proposed parallel algorithm, it is clear that the main algorithmic components can be classified as



DOI: 10.1002/fld

- **Sparse matrix-vector (SpMV)** operations for both high-order and low-order defect evaluations and multigrid grid transfers.
- **Sum reduction (SR)** for error norm estimation for use in stopping criterion of the iterative method.
- **Vector addition (VA)** that is known as an AXPY operation.
- **Thomas algorithm (TA)** for solving small sparse linear systems for line smoothing operations in the multigrid method.

This justifies focussing mainly on optimizing available hardware resources for these types of components to maximize performance.

There are two components that are not embarrassing parallel, namely, the SR and TA components. The SR operation can be implemented efficiently with $\mathcal{O}(n \log n)$ work complexity as performed in the library of parallel algorithms called THRUST [27]. The TA operation for each vertical line solve is implemented using a *blocking* technique, where the linear system is solved on a per-thread basis. This maximizes data-parallelism of the entire smoothing operation on a given grid and ensures global $\mathcal{O}(n)$ scaling of work effort for each global smoothing step. However, with this choice a current performance *akilles' heel* is that the register usage per thread is maximized and register spilling to global device memory increases when $N_z$ grows, which makes this kernel strongly memory bound. This is less of a problem for devices of CC 2.0 (fermi) where L1 caching of global memory transactions is supported, and 32K 32-bit registers are available (twice the amount available for devices of CC 1.3 (tesla)).

The SpMVs and the global smoothing step are the two key components that have the highest algorithm complexity. Both SpMV and AXPY components typically have low arithmetic intensity (ratio of computations to data-transfer) for the operations. This typically results in these kind of algorithms to be memory-bound rather than compute-bound. Therefore, we focus on optimizing for reduced data-transfers, ensuring memory coalesced read/write operations, and the use of shared memory for maximizing performance for our kernel implementation. In practice, each thread is responsible for one output element and implemented as *gather point-wise* operations in each of the SpMV and AXPY operations. In the SpMV operations, we exploit that we know the structure of the sparse matrix to avoid global assembly of matrices in compact sparse formats. Instead, the entire algorithm is implemented using *matrix-free* components to minimize the storage requirements. The matrix-free implementation is based on computing the set of possible undivided finite difference stencils on a grid with unity grid increments. This set can, then, be used for all finite difference operations under appropriate scaling when applied. The set of stencils is stored in the low-latency cached constant memory on the GPU device. This reduces both register pressure and high-latency data-transfer between streaming processors and global device memory. In the SpMV routines needed for solving the system (4) iteratively using (10), this implies that all time-dependent coefficients (5) are generated on the fly in the kernel invocations for both SpMV and smoothing operations that rely on TA. In order to speed up the SpMV operations used in the multigrid preconditioning step of the DC method, we have written separate optimized kernels where, for example, the low-order and known stencils are hard-coded as automatic (register) variables.

To avoid reduced performance due to need for conditional branching in active warps, we exert flow control by using *static branch resolution* [12] for updating the boundary points separately from the interior points using different kernels. For example, in connection with grid-transfer and smoothing operations in the multigrid algorithm, the boundary conditions imposed through fictitious ghost points are explicitly enforced after each sweep of the kernels for updating the interior points.

### *5.4. Memory storage*

Getting good performance for memory-bound applications, the key concern is to take into account the data-traffic of the algorithmic components. For implementations on GPUs, we therefore need to know the details of the memory hierarchy of the architecture. Specific to our implementation currently for Nvidia GPUs only, we use the *global memory* with latency close to 500 cycles per read/write on the device for storage of both solution and intermediate variables. For finite difference stencils, we use the low-latency read-only *constant memory* for reducing *register* pressure. For



DOI: 10.1002/fld

reducing global memory traffic whenever possible, the threads should do coalesced memory transactions and cooperate on a per-thread-block basis through low-latency (scratchpad) *shared memory*. The shared interleaved memory provides fast low-latency access when threads in active warps all access either the same or different memory banks. If this is not the case, the memory requests are serialized and performance may degrade. On the GPUs, the local memory spaces to each stream processor are a scarce resource and are shared among threads within active thread blocks scheduled for execution. Therefore, it is important to first optimize the implemented kernels to take advantage of the memory hierarchy. Then, good kernel execution configurations should be found that balance the resource available for each thread and at the same time ensure that the streaming processors can be fully occupied with computations.

To minimize data-transfer between host and device, we employ a strategy where initial data and finite difference stencil weights are initially pre-processed on the host and then transferred to the device for subsequent reuse.

The data needed to solve this three-dimensional Laplace problem (3) is the two-dimensional free surface quantities $\zeta(x, y, t)$ and $\tilde{\phi}(x, y, t)$ for a given state together with the still-water depth $h(x, y)$. The required storage is minimal compared with that of the iterative solver (10). Furthermore, by employing a $\sigma$-transformation (3) between the physical grid to a regular unity-spaced computational grid [28], the amount of data storage for the finite difference coefficients of all stencils to be used frequently in the iterative part of the solution process can be stored in the low-latency cached constant memory. This makes it possible to reduce storage requirements for the minimal set of stencils coefficients to a small array that easily fit within the 64 KB limit of the constant memory.

For the preconditioned iterative DC method (10), we need to store two fine grid arrays for the iterates together with intermediate results on the coarser grids for the multigrid preconditioning. The total storage requirement for the coarse grid levels in multigrid is bounded independently of the number of grid levels chosen by $\sum_{k=0}^{\infty} \frac{n}{s^k} = \left(\frac{s}{s-1}\right) n$, with $s$ a reduction factor indicating how aggressively the coarsening strategy is in terms of reducing the grid points for each grid level. For example, with a coarsening strategy based on simultaneous halving of grid point in the horizontal directions ($s = 4$) only between grid levels, at most $\frac{4}{3}n$ elements are needed for data storage. Dependent on the spatial resolution employed, typically, semi-coarsening is employed in the horizontal plane until the grid increments are of similar size. Thereafter, standard coarsening is employed ($s = 8$).

Results from memory scaling tests are presented in Figure 1 that shows the total amount of global device memory used as a function of problem size. The test demonstrates linear scaling in both single and double precision implementations. Furthermore, it is clear that with 4 GB RAM memory, it is possible to solve a Laplace problem of size up to close to 50/100 million degrees of freedom in single/double precision. This makes it possible to solve a system that is approximately 19 times

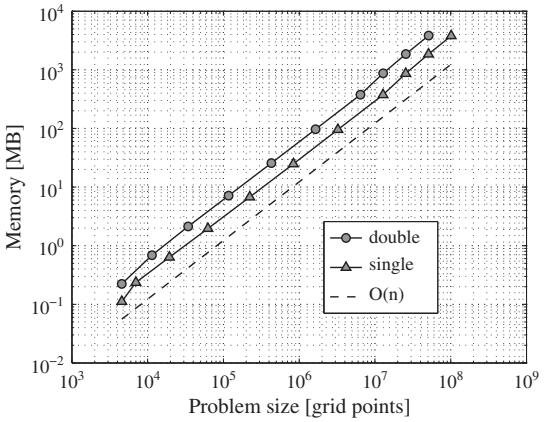



Figure 1. Scalability test of measured memory footprint for both single and double precision storage.



DOI: 10.1002/fld

larger than the large-scale validation test presented by [4] for propagating stream function waves with 4 GB RAM available.

## 6. PARALLEL PERFORMANCE ANALYSIS

For the evaluation of parallel performance of the implemented algorithms, we focus on the three characteristics of the implemented code base: (i) scalability of work effort, (ii) overall efficiency relative to a fast reference CPU code, and (iii) how well the hardware resources are utilized on the chosen hardware architecture. The performance analysis is relative to our own reference CPU implementation (to our knowledge, the best implementation available for single thread execution on a CPU). Any comparison across hardware is based on achieving the same algorithmic efficiency by executing the same algorithmic components.

The entire implementation has been subject to verification using the method of manufactured solutions, our existing CPU implementation and validation through comparison with the experiments due to [29]. As expected, the code was found to have the same performance characteristics as reported by [4, 5] in terms of accuracy (computed results) and algorithmic efficiency (iteration counts) when same algorithm and parameters are employed. The remaining issue is the overall performance of the new massively parallel implementation.

### *6.1. Test environments*

To be able to evaluate the performance of the numerical solution of (1) and (2), we have implemented the parallel algorithm described in Section 4 entirely in C++ using CUDA Toolkit v3.2 and executed in three different test environments available to us.

**Test environment 1** is based on Windows 7 (64 bit) with an Intel(R) Xeon W5590 (Intel Corporation, Santa Clara, California), 3.33 GHz CPU with 8 MB L2 RAM. The host was tested with a Quadro FX 5800 processor.

**Test environment 2** is based on Windows 7 (64 bit) with an Intel(R) Core(TM) i7 Extreme 9065, 3.2 GHz CPU with 8 MB L2 RAM. The host was tested with a GeForce GTX 480 processor.

**Test environment 3** is based on GNU/Linux Ubuntu v10.04.1 with an Intel(R) Core(TM) i7, 2.8 GHz CPU with 8 MB L2 RAM. The host was tested with a C2050 processor. We have measured the CPU-to-RAM Read/Write bandwidth to an estimated 11.5 GB/s on this work station as this is critical to the tests of the CPU algorithm used as reference in the performance comparisons.

The CUDA-enabled GPU devices we have used for our tests are listed in Table I. The host and device was in each case interconnected with a PCIe $\times 16$ Gen2 bus interface that has a theoretical peak performance of 8 GB/s bandwidth, however, with a measured effective (pinned) host-to-device transfer up to 5.5 GB/s bandwidth. This implies that using non-blocking transfer operations, computed data for the free surface state $\eta$ and $\tilde{\phi}$ at a given instant of time can be transferred to the host concurrently with device computations for post-analysis without a performance penalty. For example, if we assume that 4 GB RAM is available on the device, a maximum problem of size 50 million grid points with 6 grid points in the vertical, the surface variables can be transferred in about 0.024 s

Table I. Hardware specifications (by Nvidia) and maximum measured device-to-device bandwidth transfers for simultaneous Read/Write operations. The C2050 processor bandwidth was measured with ECC on.

| Device | Quadro FX 5800 | GeForce GTX 480 | C2050 |
|---|---|---|---|
| RAM [GB] | 4 | 1.5 | 3 |
| Cores | 240 | 480 | 448 |
| Clock rate [GHz] | 1.30 | 1.40 | 1.15 |
| Compute capability | 1.3 | 2.0 | 2.0 |
| Peak bandwidth (measured) [GB/s] | 102 (79) | 177 (154) | 144 (111) |
| Peak flop count single/double [GFLOP/s] | 624/78 | 1.344/168 | 1.030/515 |
| Release date | Nov 2008 | Mar 2010 | Mar 2010 |



DOI: 10.1002/fld

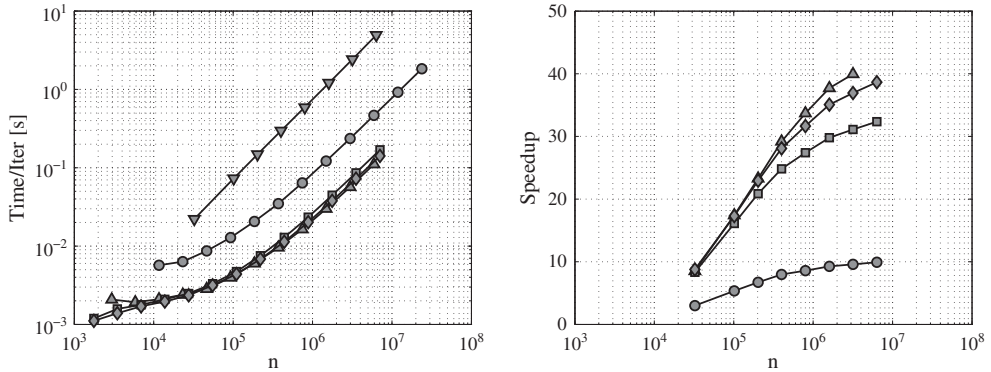
(compare with computational rates given Figure 2(a) if double precision storage is used and half of that if results are truncated to single precision).

The selected GPUs have some distinct characteristics. The Quadro FX 5800 GPU is a former and the Geforce GTX 480 a more recent high-end gaming card. The C2050 computing processor is a high-end GPU dedicated for scientific computations. These GPUs have support for double precision computations, however, the gaming cards have 1/4th the hardware support for double precision computations compared with the C2050 processor. The C2050 processor comes with software controlled Error Checking and Correction (ECC) memory (turned on) whereas the consumer cards does not. With ECC turned off, the on-chip bandwidth could be improved by close to 10%. Also, there are some noticeable relative differences in the on-chip bandwidths among the GPUs, which will be critical for application performance for memory bound parts of parallel algorithms. Thus, a direct comparison between measured CPU-to-RAM and GPU-to-RAM bandwidths suggests that we can expect 7–13 times relative speedups for these particular test environments. However, the performance of a memory bound application is dependent on the memory access patterns of individual subroutines/kernels.

### *6.2. On critical kernel optimization and analysis*

The overall performance is determined by a few critical kernels in the CPU-GPU code base developed in this work. Because the kernels are memory bound, it is critical to optimize for coalesced global memory transfers to minimize latency in order to maximize the effective bandwidth (data transferred per time unit). To assess the absolute performance of the CPU-GPU code implementation, we have profiled the code for a large enough problem size where the data-transfer have become a limiting factor to overall performance, and execution overhead is a minor fraction of total execution time. According to Figure 2, this happens when the problem size $n$ exceeds approximately 3 million grid points. We profile code execution for a problem that simulates the shoal experiment of [29] that is a standard case for both code validation and benchmarking.

A single component analysis is carried out for a test with a spatial resolution of $(N_x, Ny, N_z) = (2049, 1025, 6)$ on a domain of size $(L_x, L_y) = (35.0, 3.048)$m. We employ a sixth order accurate discretization and evaluate performance over 10-time steps with each time step $\Delta t = 0.001$s. During this test, the transformed Laplace problem is solved using the best CPU (single thread) strategy DC+MG-ZLGS-V(1,1) with error tolerances set to `atol`=$10^{-4}$ and `rtol`=$10^{-6}$. A simple breakdown of the computational expenses are given in Table II for the GPU code. In the table, the critical kernels responsible for close to 71% of total execution time have been highlighted with estimated




(a) Absolute timings.

(b) Speedup relative to CPU (singlethread) code in double precision arithmetic.

Figure 2. Scalability tests and performance comparisons in double precision arithmetic for Quadro FX 5800 (−•−), GeForce GTX 480 (−▲−), C2050 with ECC (−■−), and C2050 without ECC (−♦−) versus CPU (single thread) code (−▼−). Sixth order spatial discretization employed. Same algorithm for iterative solver DC+MG-ZLGS-V(1,1) has been employed on each architecture.



DOI: 10.1002/fld

Table II. Breakdown of performance for CPU-GPU code.

| Kernel | Operation | Time [%] | Eff. bandwidth [GB/s] | Kernel config. |
|---|---|---|---|---|
| High-order residual | SpMV+VA | 46.9 | 47.4 | $64 \times 1$ |
| ZLGS smoother | TA | 21.2 | 24.6 | $16 \times 8$ |
| Low-order residual | SpMV+VA | 3.5 | 51.9 | $32 \times 2$ |
| Sum reduction | SR | 1.6 | 98.6 | $128 \times 1$ |
| Other (aggregate sum) | – | 28.4 | – | $8 \times 8$ |
| Total | | 100 | | |

effective bandwidth achieved for the given kernel execution configurations. The kernel execution configurations have been determined by a simple *brute force* auto-tuning strategy where a larger number of configurations have been tested to find the execution configurations that gives good overall performance for a two-dimensional virtual grid of threads. The optimal execution configuration is problem-dependent, however, we have found that when then problem is sufficiently large, there are often several execution configurations that result in near-optimal performance, say, within 5% from the best configuration.

### *6.3. Benchmarking*

We benchmark the complete algorithm described in Sections 3 and 4 implemented for heterogenous CPU-GPU hardware. For benchmarking, we employ spatial discretizations of order 2, 4, and 6 for defect corrections and employ a second-order time-constant MG-RB-ZL-GS-1V(1,1) multigrid preconditioning strategy in each case. For practical computations, it is advantageous to balance the discretization parameters with the accuracy needs to best utilize the available resources.

For the performance test, we solve for the propagation of unsteady nonlinear periodic stream function solutions [30] at shallow depth (parameters for dispersion are $kh = 0.5$ and nonlinearity $H/L = 30\%(H/L)_{\max}$). To measure overall throughput performance, the choice of test case is not important as long it is numerically stable during tests.

In the performance tests, we have employed a base configuration $(N_y, N_z) = (7, 6)$. In the $x$-direction, we have used a resolution up to $N_x = 671,745$ points and the wave resolution is kept fixed close to 32 points per wave (PPW) in any test configuration. The waves propagate along the $x$-axis. At the largest resolution, it is possible to resolve a total of 20,995 wave lengths in the $x$-direction of this NWT. For the time-integration, we employ ERK4 with a Courant number $C_r = c\frac{\Delta t}{\Delta x} \approx 0.5$ that measures the ratio between resolution in time and space.

Scalability tests are presented in Figure 2(a). The absolute timings presented are suitable for comparison with performance of other models. A relative performance comparison with the algorithm executed on the CPU of test environment 3 is presented in Figure 2(b). We find that the implemented GPU model outperforms the CPU implementation for all problem sizes where $n > 2{\cdot}10^4$ in each test environment when double precision arithmetic is used. If instead single precision arithmetic is used, the throughput performance of one iteration of the iterative solver can be improved significantly because the data-storage for all variables will be halved. As illustrated in Figure 3, the performance for a single precision solver is improved up to 42% with ECC and 52% without ECC compared with a double precision solver with ECC (cf. Figure 2(a)). This improvement is achieved for the largest problem sizes only.

For the best architecture, namely the C2050 processor with ECC on, we have measured the relative sensitivity to the discretization order of accuracy as presented in Figure 4. These results show that with the second order discretization as the reference, the throughput performance is increased with up to 25% when we use a 6th order discretization instead of a 2nd order discretization. Thus, we conclude that the performance does not change dramatically with changes in the discretization order. This is attributed to that most kernels in the algorithmic strategy are memory bound and does not experiences any changes in algorithmic intensity when discretization order changes. The mapped GPU implementation achieves a speedup up to $\times 40/\times 49$ in double/single precision in comparison



DOI: 10.1002/fld

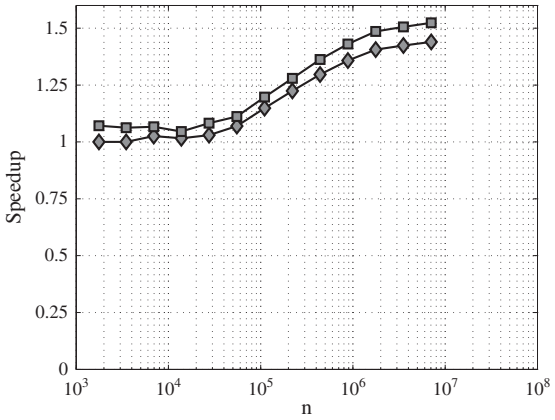



Figure 3. Speedup in scalability tests for C2050 with ECC for a single versus double precision arithmetic comparison. Single precision with ECC (−♦−) and without ECC (−■−). Iterative solver DC+MG-ZLGS-V(1,1) and sixth order spatial discretization have been employed.

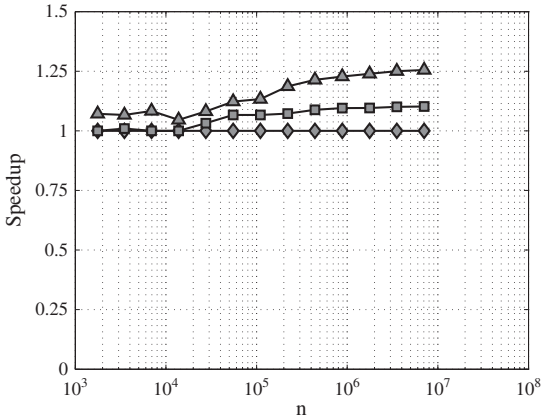



Figure 4. Speedup in scalability tests for C2050 with ECC in double precision arithmetic. Discretization is second order (reference, −♦−), fourth order (−■−), and sixth order (−▲−). Iterative solver DC+MG-ZLGS-V(1,1) has been employed.

with the existing efficient and fairly optimized reference CPU (single thread) Fortran 90 double precision code used by [4, 5]. For computations using ERK4 (with a stage count of four) and average number of iterations between 5 and 10, we find that the compute time per time step is between 20 and 40 times larger than the throughout performance per iteration measured in Figure 2(a). This implies that for $n = 10^4$, one time step is processed in about 0.04–0.08 s, for $n = 10^5$ it takes 0.08–0.16 s, and for $n = 10^6$ it takes 0.4–0.8 s of compute time. For example, we estimate that without taking accuracy into account, this enable throughput performance improvements of close to one order in magnitude reduction in comparison with the benchmarks collected and reported by [31] for a few known free surface models. The benchmarks provided in this section can be used to predict general performance based on chosen discretization parameters.

### *6.4. Performance prediction*

For practical purposes, it can be useful to use the absolute timings presented in Figure 2(a) to predict how fast we can resolve one wave period of a wave. This can be done with the following simple relationship

$$\frac{\text{Compute time}}{\text{Wave period}} \approx \frac{\text{Compute time}}{\text{Iteration}} \cdot \frac{\text{Iterations}}{\text{Time step}} \cdot \frac{\text{Time steps}}{\text{Wave period}}$$



DOI: 10.1002/fld

that can be expressed more mathematically (in terms of standard dimensionless quantities)

$$\frac{t_{compute}(n)}{T} \approx \frac{\text{Compute time}}{\text{iteration}} \cdot K, \quad K \approx \left(S_{RK} \cdot I_{avg}\right) \cdot \frac{\text{PPW}}{C_r} \tag{15}$$

where $K$ is a parameter defined in terms of the average iteration count for the solution of (9) using the DC method (10), and $S_{RK}$ is the number of stages in the Runge–Kutta ODE solver employed. If we assume that we can resolve a wave with an accuracy of less than 2% in dispersion (phase speed) if we use 10 grid points in the vertical and 10 points per wave length (e.g., see the dispersion analysis by [5]), then we find that $K \approx 4 \cdot 5 \cdot \frac{10}{0.5} = 400$. Thus, we need to multiply the absolute timings given in Figure 2(a) with 400 to estimate how fast we can resolve the propagation of the such waves in a domain of size $n$ over one wave period within 2% accuracy in dispersion.

For comparison of performance differences between free surface models, for example, Boussinesq-type and fully nonlinear free surface models, it is fair to examine how fast the solution can be advanced over a fixed time interval and at the same time satisfy the accuracy requirements (fx. dispersion accuracy). For the fully nonlinear model presented, the computational bottleneck problem is the iterative solver. This implies that the estimated timings for this model (fx. Figure 2(a)) is simply divided by $N_z$ to express the timings in terms of the number of free surface points $N_x \cdot N_y$ instead of total points $n$.

With achieved improvements in model performance, it is also tempting to ask when *real-time* analysis is feasible? This could be of interest to the coastal engineering community, however, also to other scientific areas, for example, physics-based visualization as in the game industry, ship simulator systems, real-time modeling for use in forecasting and warning systems, and fast and inexpensive numerical towing tanks experiments. Using (15), we deduce that real-time analysis is only made possible for a given problem size $n$ if all resolved waves have wave periods $T$ that fulfills $t_{compute}(n) \leqslant T$. Remark, that the resolution of the resolved waves is determined by the accuracy requirements for a given problem size $n$ with the level of accuracy of the discrete model determined by the discretization parameters. Furthermore, for many practical purposes, the parameter $K$ is typically between 150 and 600, that is, varying with a factor of four, it is clear that the main barrier for real-time analysis is the throughput performance that is ultimately determined by the hardware and programming effort put into code optimization.

For linear monochromatic waves over a flat seabed, the wave period is defined as $T = \frac{2\pi}{\sqrt{gk\tanh(kh)}}$ [32] where $k = \frac{2\pi}{l}$ is the wave number defined in terms of wave length $l$. Wave periods for steep nonlinear waves can be determined using stream function theory and are found to be smaller than those predicted by linear theory for all $kh$ under assumption of constant depth as illustrated in Figure 5. We therefore conclude that linear theory represents a conservative choice for analysis of performance for constant $kh$ and $C_r$ for the model according to (15).

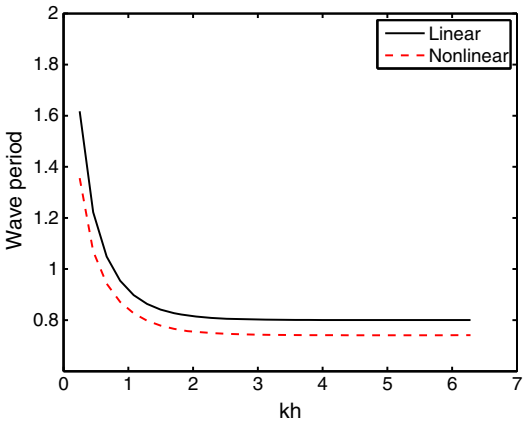



Figure 5. Comparison of wave period between linear and nonlinear stream function waves ($H/L = 90\%(H/L)_{\max}$) for a constant depth under assumption of wave length $l = 1$ m.



DOI: 10.1002/fld

To illustrate how we can estimate if real-time analysis is possible, assume that we want to simulate linear waves with wave period $T = 2$ s (fx. relative depth $kh = 0.33$ and wave length $l = 1$ m) over as large an area as possible. We further assume that we choose discretization parameters such that a typical $K \approx 400$ is achieved. By examining Figure 2(a), it can be deduced that we should expect to be able to solve problems of size $n \approx 10^5$ with the resolution of the waves fixed. This implies that faster than real-time analysis is already realistic for many simulations in two-space dimensions, for example, a NWT where approximately $10^4$ points in the horizontal and 10 points in the vertical directions are employed to resolve propagating waves. If we can satisfy our accuracy needs with $N_z = 10$ and $PPW = 50$ for primary wave frequency (e.g., see the submerged bar test of [20]) this implies that numerical experiments is made possible for a long two-dimensional NWT with close to 200 wave lengths of the primary wave resolved. At this resolution, several higher time harmonics would also be well-resolved, for example, when using spatial discretization of sixth order of accuracy. For the submerged bar test, this would be an estimated wave tank of length 480 m on the GPU that is in contrast to the 30–35 m long NWT typically used for analysis on CPU. However, if we can accept to be up to a factor $\times 10$ slower than real-time in three space dimensions then problems of size up to $n \approx 2 \cdot 10^6$ can be solved enabling even larger problems or better resolution of wave field.

## 7. CONCLUSIONS

We have carefully implemented and analyzed the performance of a baseline GPU accelerated fully nonlinear free surface potential flow model using the vendor-specific CUDA programming model. We have found that the runtime performance in double precision arithmetic could be speedup close to 40 times in comparison with our efficient CPU (single-thread) implementation based on the same algorithm. We do not claim that the presented performance results are optimal, however, the performance analysis of the implemented solver shows that fast analysis and desktop computing for coastal and offshore engineering problems is possible with modern many-core GPU hardware.

We have refrained from utilizing multi-GPUs. The immediate benefit in this choice is that we avoid transferring data between GPUs and can easier maintain the dataset on a single GPU. To compensate for this choice, we have focused on minimizing data-storage requirements to be able to solve as large problems as possible (measured in terms of the total number of grid points that can be used for the discretization of the discrete Laplace problem). We have, at this stage, focused on proof-of-concept of the choices of algorithm and technology and in future work, we will focus on domain decomposition methods for introducing more geometric flexibility into the modeling basis.

To give the work some perspective, we find that it is noticeable that the model is implemented for use with modern GPUs. These GPUs can easily work in or be moved between existing desktop computing hardware setups without affecting performance when performance is not limited by the CPU host characteristics or bus interface. Also, modifying the implemented code to instead execute via the OpenCL industry standard for better portability across hardware is considered a relatively small next step because similar syntax and functionality are provided in this standard.

The presented work is a significant step towards real-time analysis at large scales for fully nonlinear free surface simulations in coastal and offshore engineering problems.

ACKNOWLEDGEMENTS

This research used resources of GPULAB (http://gpulab.imm.dtu.dk) of Department of Informatics and Mathematical Modeling, and has been supported by grant no. 09-070032 from the Danish Research Council for Technology and Production Sciences. The GPU hardware donated by Nvidia Corporation to support the research is gratefully acknowledged.

REFERENCES

1. Cai X, Pedersen GK, Langtangen HP. A parallel multi-subdomain strategy for solving boussinesq water wave equations. *Advances in Water Resources* 2005; **28**(3):215–233.
2. Fructus D, Grue J. An explicit method for the nonlinear interaction between water waves and variable and moving bottom topography. *Journal of Computational Physics* 2007; **222**:720–739.



DOI: 10.1002/fld

3. Panchang VG, Xu B, Demirbilek Z. Wave prediction models for coastal engineering applications. In *Developments in Offshore Engineering: Wave Phenomena and Offshore Topics*, Herbich JB, Ansari KA, Chakrabarti SK, Demirbilek Z, Fenton JD, Isobe M, Kim MH, Panchang VG, Randall RE, Triantafyllou MS, Webster WC, Xu B (eds). Gulf Professional Publishing: Houston, 1999; 163–194.
4. Engsig-Karup AP, Bingham HB, Lindberg O. An efficient flexible-order model for 3D nonlinear water waves. *Journal of Computational Physics* 2009; **228**:2100–2118.
5. Engsig-Karup AP. Efficient low-storage solution of unsteady fully nonlinear water waves using a defect correction method. *Submitted to Journal of Scientific Computing* 2011.
6. Asanovic K, Bodik R, Catanzaro BC, Gebis JJ, Husbands P, Keutzer K, Patterson DA, Plishker WL, Shalf J, Williams SW, Yelick KA. The landscape of parallel computing research: A view from berkeley. *Technical Report UCB/EECS-2006-183*, EECS Department, University of California, Berkeley, December 2006.
7. Hillis WD. What is massively parralel computing, and why is it important? *Daedalus* 1992; **121**(1):1–15. (Available from: http//www.jstor.org/stable/20025415).
8. Axelsson O, Neytcheva MG. Some basic facts for efficient massively parallel computation. *CWI Quarterly* 1996; **9**:9.
9. Wallin D, Lof H, Hagersten E, Holmgren S. Reconsidering algorithms for iterative solvers in the multicore era. *International Journal of Computational Science and Engineering* 2009; **4**:270–282.
10. Patterson DA, Hennessy JL. *Computer Organization and Design: The Hardware Software Interface*, 4th ed. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2008.
11. McKee SA. Reflections on the memory wall. In *Proceedings of the 1st Conference on Computing Frontiers*, CF '04. ACM: New York, NY, USA, 2004; 162.
12. Owens JD, Luebke D, Govindaraju N, Harris M, Krüger J, Lefohn AE, Purcell TJ. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports, August 2005*, 2005; 21–51.
13. Brodtkorb A, Dyken C, Hagen T, Hjelmervik J, Storaasli O. State-of-the-art in heterogeneous computing. *Journal of Scientific Programming* 2010; **18**(1):1–33.
14. Hwu W-MW. *GPU Computing Gems*. Morgan Kaufmann: Amsterdam, Burlington, MA, 2011.
15. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with cuda. *ACM Queue: Tomorrow's Computing Today* 2008; **6**(2):40–53.
16. NVIDIA. NVIDIA CUDA C Programming Guide Version 3.2, 2010.
17. NVIDIA. CUDA C Best Practices Guide Version 3.2, 2010.
18. Li B, Fleming CA. A three dimensional multigrid model for fully nonlinear water waves. *Coastal Engineering* 1997; **30**:235–258.
19. Zakharov VE. Stability of periodic waves of finite amplitude on the surface of a deep fluid. *Journal of Applied Mechanics and Technical Physics* 1968; **9**:190–194.
20. Bingham HB, Zhang H. On the accuracy of finite-difference solutions for nonlinear water waves. *Journal of Engineering Mathematics* 2007; **58**:211–228.
21. Demmel JW, Heath MT, van der Vorst HA. Parallel numerical linear algebra. *Society for Industrial and Applied Mathematics* 1997. (Available from: http//citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.40.6056).
22. Ahern S, Alam S, Fahey M, Hartman-Baker R, Barrett R, Kendall R, Kothe D, Messer OE, Mills R, Sankaran R, Tharrington A, White JB, III. Scientific application requirements for leadership computing at the exascale. *Technical Report*, December 2007. Prepared by OAK RIDGE NATIONAL LABORATORY for the U.S. DEPARTMENT OF ENERGY under contract DE-AC05-00OR22725.
23. Trottenberg U, Oosterlee CW, Schüller A. *Multigrid*. Academic Press, 2001. ISBN 012701070X.
24. Göddeke D, Strzodka R, Mohd-Yusof J, McCormick P, Wobker H, Becker C, Turek S. Using gpus to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering* 2008; **4**(1):36–55.
25. Göddeke D. Fast and accurate finite-element multigrid solvers for PDE simulations on GPU clusters. *PhD. Thesis*, Technische Universität Dortmund, Fakultät für Mathematik, May 2010.
26. Wong H, Papadopoulou M-M, Sadoohi-Alvandi M, Moshovos A. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.
27. Hoberock J, Bell N. THRUST v1.3.0 - Library of parallel algorithms, 2011. Open source project. http://code.google.com/p/thrust/.
28. Thompson JF, Warsi ZUA, Mastin CW. *Numerical Grid Generation - Foundations and Applications*. Elsevier North-Holland, Inc.: New York, NY, USA, 1985.
29. Whalin RW. The limit of applicability of linear wave refraction theory in a convergence zone. *Research report H-71-3*, U.S. Army Corps of Engineers, WES, Vicksburg, Mi, Res. Rep, 1971.
30. Rienecker MM, Fenton JD. A fourier approximation method for steady water waves. *Journal of Fluid Mechanics* 1981; **104**:119–137.
31. Young C-C, Wu CH, Liu W-C, Kuo J-T. A higher-order non-hydrostatic $\sigma$ model for simulating non-linear refraction-diffraction of water waves. *Coastal Engineering* 2009; **56**:919–930.
32. Svendsen IA, Jonsson IG. *Hydrodynamics of Coastal Regions*. Den Private Ingeniørfond Technical University: Lyngby, eksp. København, 1976.



DOI: 10.1002/fld